Jeffrey Cocklin

2-11-17

CSCE 311 Project Threads

For my solution to the threads problems I started by implementing the simplest structures and methods followed by the more complex ones, in the order I believed to be their execution I began by creating a GenericList called readyThreads to store my Threads with status ThreadReady, and initiated under init method. Next I filled in the do\_interruptHandler method with the instructor suggestion of a call to ThreadCB’s dispatch method.

In the do\_create method I tested whether the task which would be assigned the newThread was null, and had the capacity to accept a new thread. If the state of the task allowed for an additional thread, I gave the new thread the priority of that task, and related the task to the thread and the thread to the task. If successful added to the task, the thread would then be added to the readyThreads queue, a call to dispatch would be made, and newThread would be returned. Failure of any of the aforementioned instructions would result in a call to Dispatch and a return value of null.

The next method I addressed was do\_dispatch, my solution was to write a method that would pre-empt a running thread, by determining the current thread and setting its status to ThreadWaiting, if the thread was not null. The task in control of that thread would then have its current thread set to null. Regardless of whether the threads PTBR was null or not, the PTBR would be set to null. In the event of a NullPointerException while accessing the current thread, the exception would be caught and ignored. The currentThread would then be returned to do\_dispatch. If the thread was null, it would be ignored, but in the case it was not null it would be appended to the readyThreads. A new thread, (nextThread), would then be selected off the head of the readyThreads queue, and the PTBR would be set for that thread. The thread’s status would then be changed to running, and the thread’s task would now have its current thread set to nextThread. Assuming readyThreads is not empty success is returned, otherwise failure is returned.

The Suspend method first starts by checking if the calling thread is null, if so the method does nothing. If the thread is not null, its status is then determined, ThreadWaiting or above status results in the ThreadWaiting value being incremented by one. A status of ThreadRunning is set to ThreadWaiting, and the thread’s task’s current thread is set to null. If the calling thread is contained in readyThreads, it is removed. Finally, the event that caused the suspension has the thread added to its waiting queue.

My solution for the do\_resume method follows from OSP 2 manual example on page 41. My do\_resume checks to make sure the calling thread is not null, before once again determining the status of the thread. A status below ThreadWaiting results in a return, to avoid resuming a thread that has not been suspended. A status value greater than ThreadWaiting results in a decrement to the thread’s status, moving that that thread toward a ThreadReady status. In the case the thread’s status is equal to ThreadWaiting, its status is changed to ThreadReady. Thread’s with status ThreadReady are then appended to readyThreads. Finally, a call to dispatch is made.

Inside the do\_kill method, the status of the thread is retrieved. Threads with ThreadReady status are removed from the readyThreads queue, and if the thread is running it is pre-empted. Regardless of previous status, the thread’s new status changes to ThreadKill, and said thread is removed from its task. All of the Devices which have I/O request from the thread are looped through with an instruction to have those requests cancelled. Resources held by the thread are freed and a call to dispatch is made. Finally, a count on the number of threads of the dead thread’s task is made. In the case of the count being zero the task itself is killed.